Employing Machine Learning Method in solving an Optimal Control Problem

Yicheng Feng yf2289
Mentor: Prof. Mathieu Laurière

April 15, 2025

Abstract

This Senior Thesis mainly introduces a method of using Deep-ONET Neural Operator to solve the HJB Equation of Optimal Portfolio Selection problem raised by Merton.

Contents

1 Introduction

1.1 Introduction

In this thesis, we aim to use the DeepONet Neural Operator to solve the optimal control problem proposed by Merton, which is a portfolio selection problem taken directly from *An Introduction to Mathematical Optimal Control Theory*, a set of lecture notes by Evans.

However, what exactly is the DeepONet Neural Operator? And what is an Optimal Control Problem? These two terms may sound unfamiliar to readers who have not encountered them before. This is precisely why we begin by reviewing the **Background Knowledge** relevant to this problem (Section 2).

In that section, we first examine the Optimal Control Problem in the following manner:

1. We present the basic formulation of the optimal control problem in the discrete setting.

2. We extend the formulation to the continuous setting.

3. We introduce the Hamilton–Jacobi–Bellman (HJB) equation.

These materials provide a sufficient foundation for understanding our problem.

Next, we introduce the DeepONet Neural Operator, where we:

1. Review the fundamental concepts of deep learning and neural networks.

2. Present the definition of a neural operator.

3. Describe the architecture of DeepONet.

After we go over the background Knowledge, we could introduce the problem we work on, and this is the section 3 **Problem Description**.

Now everything is ready, so it is time to actually solve this problem Numerically. And this is why we have our section 4 **Algorithm**, where we not only introduce the overview of the coding we did, but we also provide Pseudo Code for more details.

Naturally, in section 5 **Numerical Results**, we showed the result we got from the experiment. And finally, we conclude what we did and the limitations we have in section 6 **Conclusion**.

1.2 Literature Review

L. C. Evans, *An Introduction to Mathematical Optimal Control Theory*

The main source referenced in this thesis is a set of lecture notes that provides a comprehensive introduction to Optimal Control Theory — from the basic settings to concrete applications, accompanied by detailed and illustrative examples. In our thesis, we follow the logical structure presented in these notes when introducing the theory of optimal control. Moreover, the specific problem we aim to solve is taken directly from this source.

D. P. Bertsekas, *Dynamic Programming and Optimal Control*

This is a typical textbook in learning dynamic programming and optimal control theory. In our thesis, we mainly use it as a supplement of the first sources. Especially when we need to go over some specific definitions or theorems, we will check this book.

L. Lu,etc. *Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators*

This article introduces and implements the DeepONet architecture for the first time, and it also provides a Universal Approximation Theorem of DeepONet architecture as a support for DeepONet to learn nonlinear operator. For this reason, the material of this article fits for our thesis. We use it to introduce some basic ideas about Deep Learning, Neural Network, Neural Operators and DeepONet architecture, and we cite the Universal Approximation Theorem of DeepONet as a support for the feasibility of our approach.

2 Background Knowledge

2.1 Optimal Control Theory

2.1.1 Introduction

Definition 2.1 (Optimal Control). *Optimal control theory is a branch of control theory that deals with finding a control for a dynamical system over a period of time such that an objective function is optimized.*

This definition may seem to be a little bit abstract by now. Also, as a new term ,"Control" is not defined . But we will clarify this later. For now, just try to gain some basic intuitions with this primary definition.

There are many applications of optimal Control Theory: ranging from finance to engineering, including Inventory Management, Energy Scheduling, Path Planning, and Portfolio Management. The problem that will be discussed in this paper originates from Portfolio Management.

2.1.2 Discrete-Time Optimal Control Problem

Definition 2.2 (State). *The function used to describe the dynamic system's condition at each time step.*

Definition 2.3 (control). *The function used to describe the action taken at each time step.*

Remark 2.1. *The concept of control in optimal control theory is critical to guiding the system from one state to another. It represents the actions that must be taken at each moment in time to achieve the desired outcome while minimizing or maximizing the specified objective.*

Remark 2.2. *Without special clarification, we will denote the state and the control of dynamic system at each time step t as x_t and a_t. Moreover, as functions, we denote the action and control spaces as $\mathcal{X}$ and $\mathcal{A}$.*

Except for the State and Control spaces, time space is also essential in a dynamic system. Two things should be considered primarily about time space: one is whether the time is discrete or continuous, the other is whether there is a finite terminal time T. (i.e.,whether the dynamic system is finite or infinite.) For this moment, we just set up our basics under discrete time case. Furthermore, we will denote our time space as $\mathcal{T}$.

Example 2.1. *Consider a student who needs to finish its Senior Thesis in one month. So we could make the time space $\mathcal{T}$ be $[0, 30]$(days). And we make time discrete. Then x_t could represent its percentage of completion at time t (t-th day), while a_t represents how much percentage it progresses during time t (t-th day). Accordingly, $\mathcal{X}$ will have range $[0, 100]$, and $\mathcal{A}$ also ranges from $[0, 100]$.*

Definition 2.4 (Transition Function). *A function(or a sequence of functions) that returns the state of the next step for the given time's state and control.*

Remark 2.3. *We will denote the Transition function at each time step t by F_t. Notice that F is a function such that $F : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$ where:*

$$x_{t+1} = F_t(x_t, a_t)$$

And t takes discrete values on $\mathcal{T}$.

When we go to the continuous case, an updated version of the Transition function will be shown.

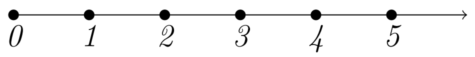
Example 2.2. *Following example 1.1, we could notice that $F_t(x_t, a_t) = x_t + a_t$.*

Remark 2.4. *The dynamics $x_{t+1} = F_t(x_t, a_t)$ actually describes an ideal case that every transition could be predicted. However, this is not what real life goes like, where some or much randomness could exist. To simulate this kind of randomness, we could add a noise term ϵ_t, which is a random variable on $\mathcal{T}$ independent with x_t and a_t.So we could update our dynamics with :*

$$x_{t+1} = F_t(x_t, a_t) + \epsilon_t$$

Definition 2.5 (deterministic and stochastic dynamics)**.** *A dynamic system is deterministic if there is no noise term in the transition. Otherwise it is stochastic.*

With all the basic settings above, we may deal with a detailed(while basic) example of Optimal Control Theory.



Example 2.3 (1-D Grid in discrete time)**.**

Consider the 1-D grid shown above. Hence, each point on the line is a state named after its number. Therefore, We have state space $\mathcal{X} = [0, 5]$ ($x = 0, 5$ are absorbing), time space $\mathcal{T} = [0, T]$, where T is the terminal time (We can set T to be infinity), and control space $\mathcal{A} = \{0, 1, -1\}$. This means that our control a_t at each time t will take value in the set {0,1,-1}.(0 means stay, 1 means moving 1 step to the right, -1 means moving 1 step to the left.)Then accordingly our dynamics will be defined by:

$$x_{t+1} = x_t + a_t$$

The case above turns out to be a totally deterministic dynamics, for instance, if we have $x_3 = 3$ meaning that we are at state 3 at time 3, and $a_3 = 1$ meaning that we choose to move 1 step to the right at time 3. Then we have a deterministic $x_4 = 4$. But we could also make this dynamics stochastic by doing the following: Let ϵ_t be a discrete random variable such that:

$$P(\epsilon_t = g) = \begin{cases} \frac{1}{3}, & \text{if } g = 1 \\ \frac{1}{3}, & \text{if } g = -1 \\ \frac{1}{3}, & \text{if } g = 0 \end{cases}$$

And then we have our new dynamics accordingly:

$$x_{t+1} = x_t + a_t + \epsilon_t$$

Since the term ϵ_t is random, so even we know that $x_3 = 3$ and $t_3 = 1$, we can not directly predict the value of x_4, instead: we have:

$$P(x_4 = g) = \begin{cases} \frac{1}{3}, & \text{if } g = 3 \\ \frac{1}{3}, & \text{if } g = 4 \\ \frac{1}{3}, & \text{if } g = 5 \end{cases}$$

Since randomness is shown in stochastic cases, it is natural that probability appears.

Definition 2.6 (Transition Probability)**.** *The probability that ends in state x' provided that one chooses control a in state x at time t. Denote it as $P_t(x' \mid x, a)$.*

Remark 2.5. *The last probability in Example 1.3 can be written in Transition Probability in the following way:*

$$P_4(3 \mid 3, 1) = P_4(4 \mid 3, 1) = P_4(5 \mid 3, 1) = \frac{1}{3}$$

By now, we have gone over the basic settings of dynamic system. It is time for us to go to optimal control problem, which deals with controls in order to minimize the **Objective Function.**

Definition 2.7 (Cost Function)**.** *The function that associates the state and control at a time step t with a real value is called cost function. We denote it by $c_t : \mathcal{X} \times \mathcal{A} \to \mathbb{R},\ t = 0, 1, 2, \ldots$*

Example 2.4 (1-D Grid, revisited)**.** *Following Example 1.3, where we have already defined the state and control function for the 1-D Grid Problem, now we may define our cost function to be:*

$$c_t(x_t, a_t) = |x_t - 3| + |a_t|$$

We often call the part $|x_t - 3|$ as state cost and the part $|a_t|$ as control cost, which are just easy to understand from the naming. This cost function means the state 3 is the "favorite" state of our dynamics hence we do not need to spend any cost at it. And the part of control cost specifies the price we need to pay for each decision made.

Remark 2.6. *The definition of cost function is abstract. It does not necessarily really mean some "cost" at certain time step, instead, it associates the state and control at a time step with the value we are interested in. For example, in a portfolio control problem, we may set our state to be the money we have, and control be the rate we distribute the money on stock with deposit. Then the cost will be defined as the income we have each day. In one word, "cost" is just a general characteristic which we care about.*

Nevertheless, it is not of great use if we only consider cost in one single step. In the problem of 1-D grid, we want to find a optimal control that could minimize the total cost. And in the portfolio control problem, we want to find a optimal control that could maximize the total "cost". Therefore, we denote the total cost by:

$$J(\mathbf{a}) = \mathbb{E} \sum_{t=0}^{+\infty} c_t(x_t, a_t)$$

where $\mathbf{a} = (a_t)_{t \geq 0}$ is the sequence of control functions.

Remark 2.7.

1. *The denotation of $J(\boldsymbol{a})$ greatly reveals the core of optimal control problem. Although x_t seems a variable in this function, but x_t is influenced and controlled by a_t. And once the initial state is anchored, the whole dynamics will be totally dependent on $\boldsymbol{a} = (a_t)_{t \geq 0}$. Therefore, by choosing different control $\boldsymbol{a}$, we will get different values in the total cost function $J(\boldsymbol{a})$.*

2. *The Expectation in $J(\boldsymbol{a}) = \mathbb{E} \sum_{t=0}^{+\infty} c_t(x_t, a_t)$ is designed for stochastic case. Remember we have introduced the definition of* ***Transition Probability.*** *Obviously, the Expectation could be dropped if the dynamics is deterministic. But there is no need for doing that since take Expectation with respect to a number makes no difference.*

3. *The value range of t here does not need to be the same as the equation above, which is from 0 to ∞. Frankly speaking, more rigorous way of writing the total cost should be :*

$$J(\boldsymbol{a}) = \mathbb{E} \sum_{t=t_0}^{T} c_t(x_t, a_t)$$

where t_0 is the starting time and T is the terminal time(Which allows to be ∞)

4. *Sometimes, we will go over our dynamics in infinity horizon. However, we sometimes consider that the decisions made at early time steps are more important than the decisions made at later time steps. In this case, we may consider using infinite horizon discounted, which is described as following: Let $\gamma \in (0,1)$ and consider $c_t(x,a) = \gamma^t c(x,a)$ for all $t \geq 0$, where c is independent of t. Then $J(\boldsymbol{a})$ is given by:*

$$J(\boldsymbol{a}) = \mathbb{E}\sum_{t=0}^{+\infty} \gamma^t c(x_t, a_t)$$

Notice that here γ is called as discounted rate. Since γ takes value in $(0,1)$, so the term γ^t shrinks with time.

Definition 2.8 (Objective Function)**.** *The function $J(\boldsymbol{a})$ defined as above, which is the total cost dependent on control $\boldsymbol{a}$, is the Objective Function of a optimal control problem.*

Remark 2.8. *The name of Objective Function obviously points out its mission and position in Optimal Control Problem. As we said before, Objective Function is the bridge that connects control we take with the information we are interested in. So with this function, it is a natural process for us to find a optimal control which minimize/maximize the **objective function**, which is also the **Objective** of the Optimal Control Problem.*

Now we have gone over the basic settings of Optimal Control Theory, and also made our goal clear– Optimizing the Objective function. Now we are going to introduce some typical ways of solving this problem. And the first one is **Bellman's Principle of Optimality**(also named as **dynamic programming principle**). But before introducing it, we need to go over another essential definition.

Definition 2.9 (Value Function)**.** *The function that gives the optimal total cost of starting at a certain state and time is called the value function, which is defined as: $V_t : \mathcal{X} \to \mathbb{R},\ t \geq 0$*

$$V_t(x) = \min_a \mathbb{E}\left[\sum_{s \geq t} c_s(x_s, a_s) \mid x_t = x\right]$$

Remark 2.9. *Notice that the value function does not necessarily be the minimal Expectation of total cost, it could also be the maximum. (That's why we* *use the word "Optimal" in our definition.) To understand this, we need to go back to the definition of cost. As we mentioned before, here cost is just a general characteristic which we care about. It could be the cost of electricity in the setting of engineering, and it could also be the income in the setting of portfolio control. In the former case, we want to minimize it, while we want to maximize the later one.*

With this genius design of Value Function, we can introduce Bellman's Principle of Optimality.

Theorem 1 (Bellman's Principle of Optimality)**.** *An optimal policy has the property that, whatever the initial state and decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*

In other words, the value function satisfies the recursive relationship:

$$V_t(x) = \min_{a \in \mathcal{A}} \mathbb{E}\left[c_t(x,a) + V_{t+1}(x_{t+1}) \mid x_t = x, a_t = a\right]$$

for all $t \in \mathcal{T}$, $x \in \mathcal{X}$, where $x_{t+1} \sim P_t(\cdot \mid x_t = x, a_t = a)$.

Remark 2.10. *In the case where we deal with the finite-horizon case, (i.e. we have a finite terminal time T), we also add the following equation into the Principle:*

$$V_{T+1} = 0, \forall x \in \mathcal{X}$$

It might be a little bit awkward that we introduce the theorem of Bellman's Principle of Optimality solely here. Hence our goal is finding the optimal control that could give us the optimal Objective function, but Bellman's Principle is a principle that provides conditions for value function. This confusion will appear no more when we go to the continuous case, where we will find that the principle is an essential part of forming the **HJB Equation**.

2.1.3 Continuous-Time Optimal Control Problem

In the section of Discrete-Time case, we have gone over the basic settings of Optimal Control Problem as well as gaining some basic intuitions. Now it is time to switch to Continuous case and handle more complex cases.

The motivation for Continuous-Time Optimal Control is very natural, since many problems in finance, engineering and scientific fields are set on the background of continuous time. For example, investment control, epidemics management a d resource allocation. Thanks to calculus, we could update our definition smoothly from Discrete-Time case to Continuous-Time case.

Definition 2.10 (State Dynamics in Continuous-Time Case)**.** *The state dynamics in Continuous-Time Case is defined by the following Stochastic Differential Equation:*

$$dX_t = b(t, X_t, a_t)\, dt + \sigma\, dW_t$$

With some given initial state X_0.

Remark 2.11.

1. *The Differential Equation is named Stochastic Differential Equation because of the term W_t, which is known as the* ***Brownian Motion****. We will not expand its definition here, what matters here is that Brownian motion as a random variable, adding randomness to the dynamics, which is similar as the ϵ_t in discrete case.*
2. *$b(t, X_t, a_t)$ here is called as* ***drift term.****, which depends on time t, state x_t and actiona_t. It is somehow a update version of our previous definition of* ***Transition Function*** *in Discrete-Time case.*
3. *σ here is called as* ***Diffusion Coefficient****, which is a constant describing the volatility of the dynamic system.*
4. *We could also write the dynamic in the following explicit way:*

$$X_t = X_0 + \int_0^t b(s, X_s, \alpha_s)\, ds + \int_0^t \sigma\, dW_s, \forall t \in \mathcal{T}$$

where X_0 is the initial state. And this form is more similar to our previous definition of Dynamics of Discrete Case.

With this continuous definition of Dynamics, we shall define our cost in continuous case.

Definition 2.11 (Cost Function-Continuous Case)**.** *The function $f : \mathcal{T} \times \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ is the cost function in continuous case.*

Remark 2.12. *The cost function here always appears in the form: $f(t, X_t, a_t)$, which returns the cost at time t with state X_t and action a_t. It is easy to detect the similarity between it and our previous cost function $c_t(X_t, a_t)$. We name a new function f here just for convenience of differentiation.*

Also, we will make update to our Objective Function accordingly.

Definition 2.12 (Objective Function, Continuous Case)**.**

$$J(\boldsymbol{a}) = \mathbb{E}\left[\int_0^\infty f(t, X_t, a_t)\, dt\right]$$

Same as the definition of discrete case, here $\boldsymbol{a}$ *denotes a sequence of controls. If the dynamics has a finite terminal time(i.e. Finite-Horizon Time Case), then the Objective Function becomes:*

$$J(\boldsymbol{a}) = \mathbb{E}\left[\int_0^T f(t, X_t, a_t)\, dt + g(T)\right]$$

where T *is the terminal time in* $\mathcal{T}$*. And* $g(T)$ *denotes the cost in the terminal time step.(i.e. terminal cost)*

In here, we often call the integral which associates with function f as **running cost**.

Remark 2.13. *We may also consider the infinite-horizon discounted here as:*

$$J(\boldsymbol{a}) = \mathbb{E}\left[\int_0^{+\infty} e^{-\rho t} f(t, X_t, a_t)\, dt\right]$$

where $\rho > 0$ *is a discount coefficient. We will see this more specifically later.*

Naturally, we want to update our value function to fit the continuous case.

Definition 2.13 (Value Function-Continuous Case)**.** *Similarly as the cost function and objective function shown above(hence that Value Function is the optimal total cost at a given state and time), we have the value function* $u(t, x) : \mathcal{T} \times \mathcal{X} \to \mathbb{R}$ *to be:*

$$u(t, x) = \min_{\boldsymbol{a}} \mathbb{E}\left[\int_t^\infty f(s, X_s, a_s)\, ds \,\middle|\, X_t = x\right]$$

Similarly, if we have a terminal time T:

$$u(t, x) = \min_{\boldsymbol{a}} \mathbb{E}\left[\int_t^T f(s, X_s, a_s)\, ds + g(X_T) \,\middle|\, X_t = x\right]$$

Remark 2.14. *Same as discrete case, the* min *here could also be replaced by* max *depending on different problem settings.*

Recall that we have introduced the Bellman's Principle of Optimality in **Theorem 1**, which is also known as **Dynamic Programming Principle**, and its continuous version is as following:

$$u(t,x) = \min_{a} \mathbb{E}\left[\int_t^{t+\Delta t} f(s, X_s, a_s)\, ds + u(t+\Delta t, X_{t+\Delta t}) \,\middle|\, X_t = x\right]$$

Thanks to this well-known principle, we could finally define the Hamilton-Jacobi-Bellman PDE, which is as known as **HJB Equation**.

2.2 HJB Equation

Definition 2.14 (Hamilton-Jacobi-Bellman (HJB) Equation)**.** *Consider the dynamics in the form of SDE which we discussed before as*

$$dX_t = b(t, X_t, a_t)\, dt + \sigma\, dW_t,$$

and the Objective function accordingly

$$J(\boldsymbol{a}) = \mathbb{E}\left[\int_t^T f(s, X_s, a_s)\, ds + g(X_T)\right].$$

The value function is defined as

$$u(t,x) = \min_{\boldsymbol{a}} \mathbb{E}\left[\int_t^T f(s, X_s, a_s)\, ds + g(X_T) \mid X_t = x\right].$$

Then the value function $u(t,x)$ satisfies the Hamilton-Jacobi-Bellman (HJB) equation:

$$\frac{\partial u}{\partial t}(t,x) + \min_{\boldsymbol{a}\in\mathcal{A}}\left\{ f(t,x,a) + b(t,x,a)\frac{\partial u}{\partial x}(t,x) + \frac{1}{2}\sigma^2 \frac{\partial^2 u}{\partial x^2}(t,x)\right\} = 0,$$

with terminal condition $u(T,x) = g(x)$.

Remark 2.15.
1. *Notice that HJB Equation is a second-order nonlinear PDE about value function $u(x,t)$. And it is a necessary condition for optimality of the given Optimal Continuous-Time Control Problem. By solving HJB Equation, we could get the form of value function $u(t,x)$, and by plugging this back to the PDE, we could recover the optimal control $\boldsymbol{a}$ we need.*

2. *Since the HJB Equation Characterizes the Value Function, so the* min *could be replaced by* max *depending on different problem settings.*

Remark 2.16. *The Hamilton-Jacobi-Bellman (HJB) equation also applies to infinite-horizon optimal control problems, especially when the Objective Function is discounted.*

Assume the Objective Function takes the form

$$J(\boldsymbol{a}) = \mathbb{E}\left[\int_0^\infty e^{-\rho t} f(X_t, a_t)\, dt\right],$$

where $\rho > 0$ *is the discount rate and the system dynamics are time-invariant:*

$$dX_t = b(X_t, a_t)\, dt + \sigma\, dW_t.$$

Under suitable assumptions, the value function becomes time-independent, i.e., $u(t, x) = u(x)$*, and the HJB equation reduces to the stationary form:*

$$\rho u(x) = \min_{\boldsymbol{a} \in \mathcal{A}} \left\{ f(x, \alpha) + b(x, \alpha) u'(x) + \frac{1}{2}\sigma^2(x, \alpha) u''(x) \right\}.$$

This stationary equation characterizes the optimal value function and is used to derive optimal feedback control laws in infinite-horizon settings.

This remark calls back our precious Remark 1.13.

2.3 DeepONet Neural Operator

2.3.1 Neural Networks

Neural networks are powerful function approximators that have been widely applied in fields such as computer vision, natural language processing, and scientific computing.

Neural Networks are a very complicated class, and one we will mainly use in this thesis will be feedforward neural network.

Definition 2.15 (Feedforward Neural Network). *A standard feedforward neural network defines a mapping* $f_\theta : \mathbb{R}^n \to \mathbb{R}^m$*, parameterized by weights and biases* θ*(here* θ *denotes the collection of all parameters in the neural network), through compositions activation functions. Formally, for input* $x \in \mathbb{R}^n$*, a neural network with L layers can be written as:*

$$f_\theta(x) = W_L \sigma(W_{L-1} \sigma(\cdots \sigma(W_1 x + b_1) \cdots) + b_{L-1}) + b_L,$$

where W_i and b_i are the weight matrices and biases of the i-th dimension, and $\sigma(\cdot)$ is an activation function such as ReLU or sigmoid. According to the ***universal approximation theorem****, neural networks can approximate any continuous function on compact domains to arbitrary accuracy, given sufficient width and depth.*

Remark 2.17. *The Universal Approximation Theorem of Neural Network is really essential in Neural Network, which we will not go over here, but you are free to check it in the paper I cite in the end.*

However, traditional neural networks are designed to approximate mappings between finite-dimensional Euclidean spaces. In many scientific problems, such as the solution of partial differential equations (PDEs), the objects of interest are operators, motivating the need for neural networks that can approximate mappings between function spaces.

Under this motivation, we will introduce a kind of neural network which could approximate the mappings from function to function, which is called neural operator.

Definition 2.16 (Neural Operator)**.** *Neural operators are a class of machine learning models designed to approximate mappings between function spaces. Instead of learning a function $f : \mathbb{R}^n \to \mathbb{R}^m$, neural operators aim to learn an operator $\mathcal{G}$ that maps a function u to another function $v = \mathcal{G}(u)$, i.e.,*

$$\mathcal{G} : \mathcal{U} \to \mathcal{V}, \quad \textit{with} \quad v(x) = \mathcal{G}(u)(x),$$

where $\mathcal{U}$ and $\mathcal{V}$ are spaces of functions.

This formulation is particularly well-suited to solving families of PDEs, where the input u may represent a varying coefficient, boundary condition, or forcing term, and the output v is the corresponding PDE solution. In contrast to classical numerical solvers that compute solutions case by case (e.g., finite element or finite difference methods), neural operators aim to learn the entire solution operator over a distribution of inputs. Once trained, they can provide fast and mesh-free predictions for new inputs, making them attractive for real-time or large-scale simulation tasks.

2.3.2 DeepONet

DeepONet is a specific architecture of neural operators, firstly introduced in *Learning nonlinear operators via DeepONet based on the universal approxi-*

mation theorem of operators. by Lu, L., Jin, P., Pang, G., Zhang, Z., & Karniadakis, G. E. (2021) (Details see final reference)

Definition 2.17 (DeepONet)**.** *The DeepONet architecture consists of two subnetworks:*

- ***Branch network:*** *Takes as input a discretized version of the input function u, typically evaluated at a fixed set of sensor locations $\{x_i\}_{i=1}^m$, and outputs a feature vector $b(u) \in \mathbb{R}^p$.*
- ***Trunk network:*** *Takes as input the evaluation location $x \in \Omega$ and outputs another feature vector $t(x) \in \mathbb{R}^p$.*

The final output is obtained by taking the inner product of the two feature vectors:

$$\mathcal{G}(u)(x) \approx \sum_{i=1}^{p} b_i(u) \cdot t_i(x).$$

DeepONet has been proven to be a universal approximator for nonlinear continuous operators under mild assumptions, which is supported by the following theorem:

Theorem 2 (Universal Approximation Theorem for Operator)**.** *Suppose that σ is a continuous non-polynomial function, X is a Banach Space, $K_1 \subset X$, $K_2 \subset \mathbb{R}^d$ are two compact sets in X and $\mathbb{R}^d$, respectively, V is a compact set in $C(K_1)$, G is a nonlinear continuous operator, which maps V into $C(K_2)$. Then for any $\epsilon > 0$, there are positive integers n, p, m, constants c_i^k, ξ_{ij}^k, θ_i^k, $\zeta_k \in \mathbb{R}$, $w_k \in \mathbb{R}^d$, $x_j \in K_1$,*

$$i = 1, \ldots, n, \quad k = 1, \ldots, p, \quad j = 1, \ldots, m,$$

such that

$$\left| G(u)(y) - \sum_{k=1}^{p} \sum_{i=1}^{n} c_i^k \, \sigma \left(\sum_{j=1}^{m} \xi_{ij}^k u(x_j) + \theta_i^k \right) \sigma(w_k \cdot y + \zeta_k) \right| < \epsilon \tag{1}$$

holds for all $u \in V$ and $y \in K_2$.

In this thesis, we shall employ DeepONet to approximate the solution operator of the HJB equation arising in the given Optimal Control Problem.

3 Problem Description

After going through basic knowledge , we now consider a typical problem of Optimal Control Problem: the Merton portfolio Control problem. This Problem comes from *An introduction to mathematical optimal control theory.* By Evans. In this setting, an agent aims to optimally allocate wealth between a risk-free bond and a risky stock, while simultaneously deciding the optimal rate of consumption over time.

3.1 Optimal Control Problem Overview

3.1.1 Model Setup

Let $X(t)$ (i.e. the state) denote the wealth of the agent at time t. The agent can choose:

- $\alpha^1(t) \in [0,1]$: the fraction of wealth invested in the risky asset,
- $\alpha^2(t) \geq 0$: the rate at which the agent consumes wealth.

Notice: $(\alpha^1(t), \alpha^2(t))$ denotes the control at each time t.

The bond (risk-free asset) and stock (risky asset) follow the **dynamics**:

$$\begin{aligned} db(t) &= rb(t)\,dt, \\ dS(t) &= RS(t)\,dt + \sigma S(t)\,dW(t), \end{aligned}$$

where $r > 0$ is the interest rate, $R > r$ is the expected return of the risky asset, and $\sigma > 0$ is the volatility(What we called diffusion coefficient before).

The wealth $X(t)$ evolves according to the stochastic differential equation:

$$dX(t) = \Big[(1-\alpha^1(t))X(t)r + \alpha^1(t)X(t)R - \alpha^2(t)\Big]\,dt + \alpha^1(t)X(t)\sigma\,dW(t).$$

Noice that this is the dynamics of state as we defined before.

3.1.2 Objective Function

Let

$$Q := \{(x,t) \mid 0 \leq t \leq T,\ x \geq 0\}$$

and denote by τ the (random) first time $X(\cdot)$ leaves Q. The agent seeks to maximize the expected total discounted utility of consumption over a finite or infinite time horizon. The control problem is to maximize:

$$J(\alpha^1, \alpha^2) = \mathbb{E}\left(\int_t^{\tau} e^{-\rho s} F(\alpha^2(s))\, ds\right),$$

which is the so-called Objective Function, where $\rho > 0$ is the discount rate and $F(\cdot)$ is a utility function, typically assumed to be $F(\alpha) = \alpha^\gamma$, with $0 < \gamma < 1$.

Remark 3.1. *Here Q is the space of state and control. And the definition of it provides by the Evan's Note. With this definition, we could see that this problem turns out to be set in finite-horizon time, since τ has the upper bound T.*

3.1.3 HJB Equation

By applying the dynamic programming principle, the corresponding Hamilton–Jacobi–Bellman (HJB) equation for the value function $u(t,x)$ is:

$$\frac{\partial u}{\partial t}(t,x) + \max_{0\le\alpha^1\le 1,\, \alpha^2\ge 0}\left\{\frac{1}{2}(\alpha^1 x\sigma)^2\frac{\partial^2 u}{\partial x^2}(t,x) + \left[(1-\alpha^1)xr + \alpha^1 xR - \alpha^2\right]\frac{\partial u}{\partial x}(t,x) + e^{-\rho t}F(\alpha^2)\right\} = 0.$$

With Boundary Conditions $u(x,T) = 0$ and $u(0,t) = 0$.

3.2 Goal of our thesis

Overall, our goal is to solve this HJB Equation, (i.e. solve value function u. In Evan's Notes, one explicit solution is already provided. What we are going to do in this thesis is to employ DeepONet Neural Operator to solve the HJB Equation.

4 Algorithm

In this section, we will introduce our algorithm which learns a DeepONet Neural Operator to solve the HJB Equation. We will firstly go over the algorithm, and then we will show the pseudo code.

4.1 Overview of Algorithm

Formally Speaking, there is a standard way of solving a PDE by DeepONet Neural Operator whose central idea is to approximate the solution operator of a PDE, which maps a function in PDE (such as source terms or boundary conditions) to the corresponding solution function.

4.1.1 Standard DeepONet Framework for Solving PDEs

Let the PDE be defined over a spatial domain $\Omega \subset \mathbb{R}^d$, with the general form:

$$\mathcal{L}_u(x; a) = 0, \quad x \in \Omega,$$

where:

- $\mathcal{L}_u$ is a differential operator acting on the unknown solution u,
- $a \in \mathcal{A}$ represents input functions (e.g.source terms or initial/boundary conditions),
- The goal is to learn the mapping $a \mapsto u(\cdot)$, i.e., the solution operator $\mathcal{G}(a) = u$.

In the DeepONet framework, the solution operator $\mathcal{G}$ is approximated by a neural operator of the form:

$$\mathcal{G}_\theta(a)(x) \approx \sum_{i=1}^{p} b_i(a) \cdot t_i(x),$$

where:

- The **branch network** takes as input the function a, typically represented by its evaluations at fixed sensor locations $\{x_j\}_{j=1}^m$, and outputs $b(a) \in \mathbb{R}^p$,
- The **trunk network** takes as input the evaluation point $x \in \Omega$ and outputs $t(x) \in \mathbb{R}^p$,
- The dot product of the two yields the approximate value of the solution $u(x)$.

To train the DeepONet, one generates a dataset of input-output function pairs $\{(a^{(i)}, u^{(i)})\}_{i=1}^{N}$, where each $u^{(i)}$ solves the PDE for a corresponding input $a^{(i)}$. The loss function is typically defined by the mean-squared error (MSE) between the predicted and true solutions:

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\int_{\Omega}\left|\mathcal{G}_\theta(a^{(i)})(x) - u^{(i)}(x)\right|^2 dx.$$

Once trained, the DeepONet provides a fast and accurate approximation to the solution operator, allowing real-time evaluation of the PDE solution for new input functions a.

4.1.2 Our Settings

Generally speaking, our settings of Algorithm basically follows the standard setting. However, in order to obtain the loss function, which is the MSE between the Neural Network and the true solution, we need the true solution at first. Normally speaking, we will firstly solve the Equation by finite-difference method in order to obtain a true solution. Then we could generate training data based on that. But in our Algorithm, we will just take the explicit solution provided by the Evans notes which is the following:

$$u(x,t) = g(t)x^\gamma,$$

where:

$$g(t) = e^{-\rho t}\left[\frac{1-\gamma}{\rho - \nu\gamma}\left(1 - e^{-\frac{(\rho-\nu\gamma)(T-t)}{1-\gamma}}\right)\right]^{1-\gamma}.$$

and

$$\nu := \frac{(R-r)^2}{2\sigma^2(1-\gamma)} + r.$$

Let's recall the HJB Equation we want to solve:

$$\frac{\partial u}{\partial t}(t,x) + \max_{0\le\alpha^1\le 1,\,\alpha^2\ge 0}\left\{\frac{1}{2}(\alpha^1 x\sigma)^2\frac{\partial^2 u}{\partial x^2}(t,x) + \left[(1-\alpha^1)xr + \alpha^1 xR - \alpha^2\right]\frac{\partial u}{\partial x}(t,x) + e^{-\rho t}F(\alpha^2)\right\} = 0.$$

With Boundary Conditions $u(x,T) = 0$ and $u(0,t) = 0$. And our goal is to approximate the operator which maps function F to u by training the DeepONet Neural Network. But hence that, we are going to learn the true solution that provided by the paper, so we will take the form of function F just as the form in Evan's Note which is:

$$F(a) = a^{\gamma} \qquad (0 < \gamma < 1).$$

Generally speaking, our procedure has the following steps:

- **Data Generation** In this part, we will generate the data which is needed for training. Including sensor vector (*F_sensor*), query vector (*query_points*)and true solution (*u_true*).
- **DeepONet Class** In this part, we will define a Neural Network that helps us to learn the operator.
- **DeepONet Training** In this part, we will train our DeepONet class to approximate the operator.

Now let's go to details of these procedures by looking at the Pseudo Code.

4.2 Pseudo Code

4.2.1 Algo1-Data Generation

Algorithm 1 generate_data_with_grf

```
 1: Input: Batch size $B$, number of sensors $n_s$, queries $n_q$, time horizon $T$, etc.
 2: Output: Sensor values $F$, query points $(x, t)$, true solution $u$
 3: Sample $\gamma \sim \text{Uniform}(0, 1)$ of shape $(B, 1)$
 4: Create shared time grid $t_{\text{sensor}} \leftarrow \text{linspace}(0, T, n_s)$
 5: Repeat $t_{\text{sensor}}$ for each batch
 6: Generate GRF samples: $a_2 \leftarrow \texttt{sample\_grf}(t_{\text{sensor}}, \ell, B)$
 7: Define $F = |a_2|^{\gamma}$ (element-wise power)
 8: Sample query points:
        • $t_{\text{query}} \sim \text{Uniform}(0, T)$
        • $x_{\text{query}} \sim \text{Uniform}(0, X)$
        • query_points $\leftarrow (x_{\text{query}}, t_{\text{query}})$
 9: Compute true solution $u \leftarrow \texttt{true\_solution}(\text{query\_points}, \gamma, \rho, r, R, \sigma, T)$
10: return $(F, \text{query\_points}, u)$
```

Supporting Functions

RBF Kernel:

$$\texttt{rbf\_kernel}(x_1, x_2, \ell) = \exp\left(-\frac{\|x_1 - x_2\|^2}{2\ell^2}\right)$$

Covariance Matrix:

$$\texttt{covariance\_matrix}(x, \ell) = K \in \mathbb{R}^{N\times N}, \quad K_{ij} = \texttt{rbf\_kernel}(x_i, x_j, \ell)$$

Sample GRF:

$$\texttt{sample\_grf}(x, \ell, B) = \text{Draw } B \text{ samples from } \mathcal{N}(0, K), \quad K = \texttt{covariance\_matrix}(x, \ell)$$

True Solution Formula

Let input points be x and t, and $\gamma \in (0, 1)$.

$$\begin{aligned} \nu &= \frac{(R-r)^2}{2\sigma^2(1-\gamma)} + r \\ \text{Exponent} &= -\frac{(\rho - \nu\gamma)(T-t)}{1-\gamma} \\ \text{Then} \quad u(x,t) &= e^{-\rho t}\left(\frac{1-\gamma}{\rho - \nu\gamma}\right)^{1-\gamma}\left(1 - e^{\text{Exponent}}\right)^{1-\gamma} x^{\gamma} \end{aligned}$$

4.2.2 Algo2-DeepONet Class

Algorithm 2 DeepONet (Neural Operator Model)

```
 1: Input: Number of sensors $n_{\text{sensor}}$, latent dimension $p$, hidden size $h$, final time $T$
 2: Output: DeepONet model that approximates $u(x,t)$
 3: Store final time $T$ to enforce terminal boundary condition
 4: Define branch network:
        $\texttt{branch\_net} :=$ MLP with layers: $n_{\text{sensor}} \to h \to h \to p$
 5: Define trunk network:
        $\texttt{trunk\_net} :=$ MLP with layers: $2 \to h \to h \to p$
 6: function FORWARD(F_sensor, query_points)
 7:     Input: Sensor values $F_{\text{sensor}} \in \mathbb{R}^{B \times n_{\text{sensor}}}$, query points $(x,t) \in \mathbb{R}^{n_q \times 2}$
 8:     Output: Predicted solution $u_{\text{pred}} \in \mathbb{R}^{B \times n_q}$
 9:     Compute branch output: $b \leftarrow \texttt{branch\_net}(F_{\text{sensor}})$
10:     Compute trunk output: $t \leftarrow \texttt{trunk\_net}(query\_points)$
11:     Compute inner product: $I = b \cdot t^\top$                    ▷ Shape: $(B, n_q)$
12:     Extract spatial and temporal coordinates: $x \leftarrow query\_points[:,0]$, $t \leftarrow query\_points[:,1]$
13:     Compute boundary multiplier: $m = x \cdot (T - t)$
14:     Reshape multiplier: $m \leftarrow m^\top$ to shape $(1, n_q)$
15:     Enforce boundary: $u_{\text{pred}} = I \cdot m$
16:     return $u_{\text{pred}}$
17: end function
```

4.2.3 Algo3-Training Function

Algorithm 3 train_DeepOnet

```
1: Input: Batch size $B$, number of sensors $n_s$, query points $n_q$, final time $T$, max wealth $X$, parameters $\rho$, $r$, $R$, $\sigma$, GRF length scale $\ell$, learning rate $\eta$, number of epochs $N_{\text{epochs}}$
2: Output: Trained DeepONet model
3: Initialize DeepONet model with $n_s$, output dimension $p = 100$, hidden dimension 256, and terminal time $T$
4: Initialize Adam optimizer with learning rate $\eta$
5: Define loss function: mean squared error (MSE)
6: for epoch = 1 to $N_{\text{epochs}}$ do
7:     Set model to training mode
8:     Generate training data:
       $(F_{\text{sensor}}, \text{query\_points}, u_{\text{true}}) \leftarrow \texttt{generate\_data\_with\_grf}(B, n_s, n_q, T, X, \rho, r, R, \sigma, \ell)$
9:     Predict output: $u_{\text{pred}} \leftarrow \texttt{model}(F_{\text{sensor}}, \text{query\_points})$
10:    Compute loss: loss $\leftarrow \texttt{MSE}(u_{\text{pred}}, u_{\text{true}})$
11:    Zero gradients: optimizer.zero_grad()
12:    Backpropagate: loss.backward()
13:    Update model: optimizer.step()
14:    if epoch mod 100 = 0 then
15:        Print training progress and current loss
16:    end if
17: end for
18: return model
```

5 Numerical Results

In this section, we will go over our process of experiments. Including the hinders we confront and the the specific case we solved.

5.1 Hinders

Generally speaking, our hinders came from two major reasons:

True Solution

Let's recall the true solution of this HJB Equation :

$$u(x,t) = g(t)x^{\gamma},$$

where:

$$g(t) = e^{-\rho t}\left[\frac{1-\gamma}{\rho-\nu\gamma}\left(1-e^{-\frac{(\rho-\nu\gamma)(T-t)}{1-\gamma}}\right)\right]^{1-\gamma}.$$

and

$$\nu := \frac{(R-r)^2}{2\sigma^2(1-\gamma)} + r.$$

$R, r, \gamma, \sigma, \rho$ are parameters. And t and x are in range of $[0,T]$ and $[0,X]$, where X and T could be chosen by us.

In particular, the range of γ is given by the problem which is $(0,1)$. So originally, our code is strictly follow this, (you can see details in our pseudo code), where we randomly sample γ in $(0,1)$. But then the first problem appears: that the term $1-\gamma$ will return us "not a number" if γ is too close to 1.

To solve this problem, we fixed our γ to be 0.5 in order to avoid extreme value. But new issue shows up: **the exponential term**.

As we could observe in the true solution, there are two exponential terms which are : $e^{-\rho t}$ and $e^{-\frac{(\rho-\nu\gamma)(T-t)}{1-\gamma}}$. Since the increasing speed of exponential function is quite crazy, it will be easy to cause the value explosion if the power of exponential is large.

DeepONet Class

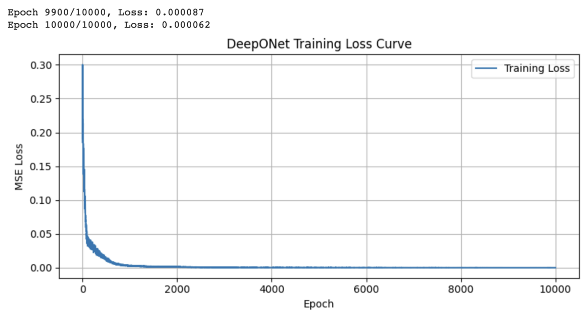
Although our DeepONet Class(Details see Pseudo code) is strong and powerful to mimic the Neural Operator, the MSE loss between the DeepONet class and the true solution is easy to explode given that the true solution is so complicated.

5.2 Numerical result in one particular setting

Originally, what we trying to do is to find solutions in different problem settings with random γ sampled. However, due to the complexity of the problem and time issue, what we finally did is solve this euqation by DeepOnet in one particular setting which is :

$$((\gamma, r, R, \sigma, \rho, T, X) = (0.5, 0.01, 0.05, 0.4, 0.2, 1, 1))$$

And finally, we have our Experimental result in the following:



```
Epoch 9900/10000, Loss: 0.000087
Epoch 10000/10000, Loss: 0.000062
```

Figure 1: Training loss curve of DeepONet over 10,000 epochs.

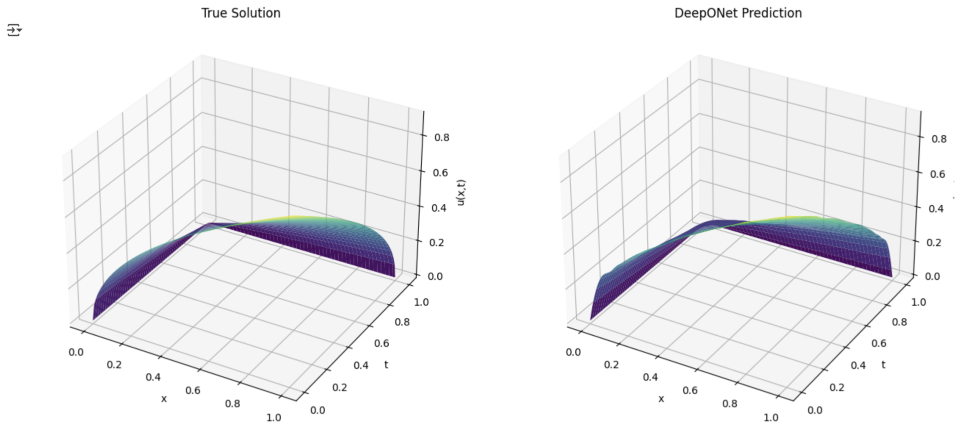


Figure 2: Comparison of the true solution and DeepONet prediction.

In Figure1, we have shown the plotting of the loss during the training over 10000 epochs. What we can see here is that finally the loss goes very low(0.000062) after 100000 epochs, which reflect the similarity between the true solution and the solution learned by us.

And what is shown in Figure 2, which is the 3-D plooting of the true solution versus the DeepONet Solution, could justify the similarity in a more visible way.

6 Conclusion

By successfully solving the HJB Equation by DeepONet Neural Network in a mild problem setting, we justify the feasibility of employing DeepONet Neural network to solve some complicated PDE.

However, some drawbacks and limitations of this method have also been observed. Firstly, it is sometimes hard for the computer to calculate the output of complicated functions. Secondly, in order to learn a complicated solution, more powerful architectures of neural network are needed. Finally, time issue of training is also an implicit restriction that could not be overlooked.

References

[1] D. P. Bertsekas, *Dynamic Programming and Optimal Control*, Volumes I and II, 4th Edition, Athena Scientific, 2017.

[2] L. C. Evans, *An Introduction to Mathematical Optimal Control Theory*, Lecture Notes, University of California, Department of Mathematics, Berkeley, 2005.

[3] L. Lu, P. Jin, G. Pang, Z. Zhang, and G. E. Karniadakis, *Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators*, Nature Machine Intelligence, 3(3), 218–229, 2021.